６．Navigation Stack

解説・調整手順書

この手順書はNavigation Stackを利用・調整するために必要な知識と手順を解説するものです。

H29モデルのパラメータを調整する場合は２．から読んでください。

ロボットの起動手順は「ロボットアプリ立ち上げ手順書」を読んでください。

新たにNavigation Stackを使う準備をする場合ははじめから読んでください。

ROSのインストールを済ませていない場合は，先に「ROS導入手順書」を読んでください。

目次は

０．ROSの基礎知識を習得する（ROS初心者の場合）

１．Navigation Stackを自作ロボットで動かしてみる

２．パラメータを調整する

３．実機で動かす設定をする（ros\_controlを理解する）

４．FAQ

です。

また，本書を読む際は接客ロボットに使った**rostestパッケージ**を参考にしてください。

1. **ROSの基礎知識を習得する（ROS初心者の場合）**

書籍がある場合は読むのもいいですが，その書籍のROSのバージョンは古いかもしれません！

一番信頼性が高いのは公式サイトのチュートリアルです。

Tutorials（日本語）

http://wiki.ros.org/ja/ROS/Tutorials

このチュートリアルの**１．初級**を読んで進めれば十分です。

1. **Navigation Stackを自作ロボットで動かしてみる**

ROSで始めるロボティクス(1)～(8)

http://bril-tech.blogspot.jp/2016

このチュートリアルを読んで進めてください。株式会社ブリリアントサービス様が提供している非常に有用なページです。実機はなくても大丈夫です。(1)～(8)まで読めばgazebo上でNavigationが動きます。

接客ロボットで使った**rostest**パッケージも上記チュートリアルに従って基本形を作りました。（構成を少し変えていますが，ほぼ一緒です）

上記サイトでは細かい原理の説明を省いて、とにかく動かすことを目標に書かれています。

Navigation Stackは以下のような構成で動いていることを知って置いてください。

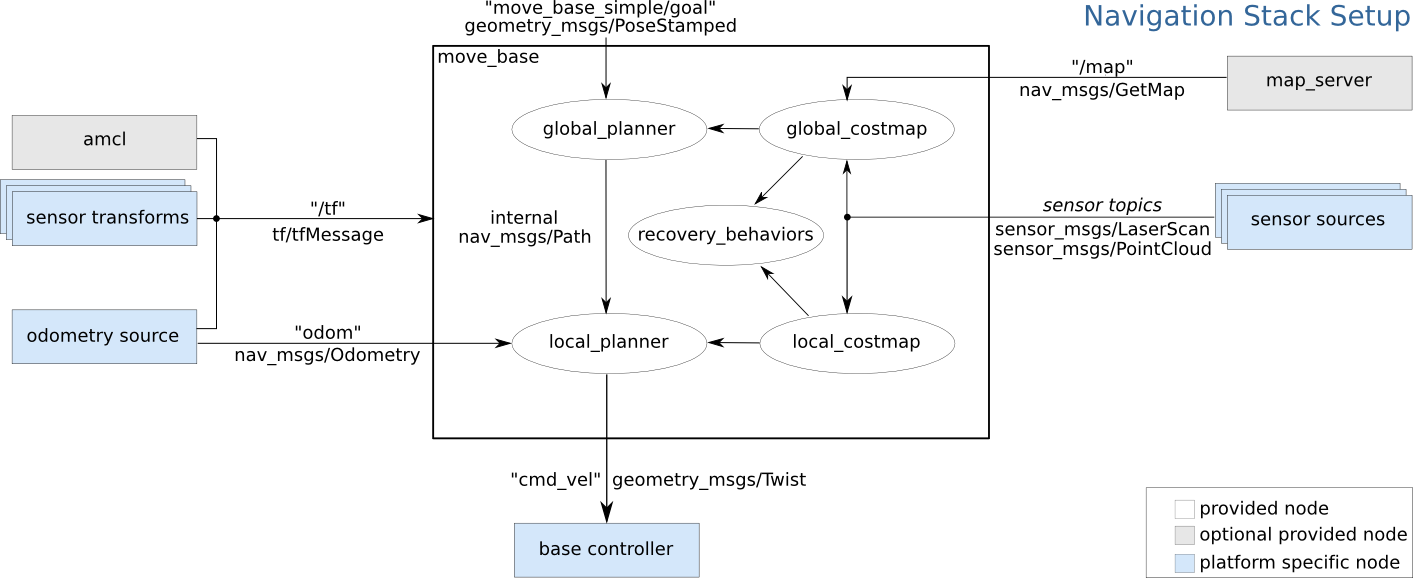


図6-1 Navigation Stack

日本語化すると「図6-2 Navigation Stack 日本語版」となります。



図6-2 Navigation Stack 日本語版

細かい原理については「３．実機で動かす設定をする」で解説しますが，原理を理解していなくても動きを改善することはできますので，先にパラメータ調整をします。

1. **パラメータを調整する**

上記サイトでNavigationが動くところまでは行きました。

その後何を調整すればいいかが難しいと思います。

パラメータのチューニングについては

Basic Navigation Tuning Guide

<http://wiki.ros.org/navigation/Tutorials/Navigation%20Tuning%20Guide>

を読むべきです。（英語です）

ROSのナビゲーションmove\_baseについて理解を深めてみる

<http://blog-sk.com/ros/ros-move_base/>

ROSのナビゲーションamclについて理解を深めてみる

<http://blog-sk.com/ros/ros-amcl/>

も，move\_baseとamclのパラメータについて日本語で解説している貴重なサイトですので参考にしてください。

本格的に調整するなら上記3つは必須なのですが，最初は難しいと思いますので，私なりの言葉で補足しながら説明します。

パラメータは大きく分けて4つあります。

(1). **ロボットの物理パラメータ**（my\_robo\_description/my\_robo.urdf）

(2). **コントローラパラメータ**(my\_robo\_control/config/controller.yaml)

(3). **move\_baseパラメータ**(my\_robo\_2dnav/\*\*\*.yaml)

rostestではmy\_robo\_2dnav/params/\*\*\*.yaml)

(4). **amclパラメータ**(my\_robo\_2dnav/launch/move\_base.launch

rostestではmy\_robo\_2dnav/params/amcl\_params.yaml)

実機で動かす場合には，さらに

(5). **YP-Spurパラメータ**(robot\_launcher/ypspur\_param/ yp-h29ki-3600-1.param)

も加わります。

このうち，Navigation Stackに関係するパラメータは

・move\_baseパラメータ

・amclパラメータ

です。しかし，実行時に発生する問題の大部分は

・ロボットの物理パラメータ

・コントローラパラメータ

にあります。つまり問題は**Navigation Stackとは関係ない部分にある**ということです。最初に遭遇する問題は，**オドメトリのずれ**でしょう。まずはここから調整します。（調整の順番を知っておくだけでもだいぶ効率が違うと思います）

また，ファイルパスはrostestでのパスを示しますので，チュートリアルとは違う場所にあることをご了承ください。（大きくは変わりません）

2.1 ロボットの物理パラメータが正しいことを確認する

物理パラメータが間違っていると，正しくシミュレートされません。

my\_robo\_descripsion/my\_robo.urdfがロボット形状を定義しているファイルです。チュートリアルそのままでやる場合は正しく設定されていますので変更する必要がありませんが，チュートリアルからロボット形状を変更することがほとんどでしょう。接客ロボットのurdfは図6-3のような見た目にしました。

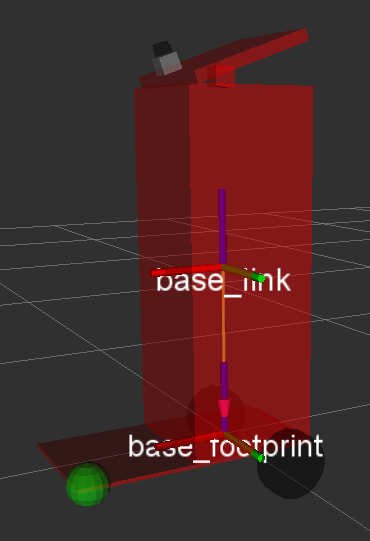


図6-3 my\_robo.urdf

まず，inertiaの正しい計算式を知っておいてください。ロボットの形状が変更されるたびにinertiaの計算が必要になります。inertiaは3×3回転慣性行列ですが，回転慣性行列は対称なので，属性ixx、ixy、ixz、iyy、iyz、izzを使用して、この行列の6つの対角要素のみを指定します。**このうちixy, ixz, iyzは，直方体，円柱，球の場合は０でいいです。**

（例）

奥行きd,幅w,高さh,重さmの直方体のinertiaは

|  |
| --- |
| ixx = 1.0/12.0\*m\*(w\*w+h\*h)  iyy = 1.0/12.0\*m\*(h\*h+d\*d)  izz = 1.0/12.0\*m\*(w\*w+d\*d) |

半径r, 長さl,重さmの円柱の場合，

|  |
| --- |
| ixx = 1.0/12.0\*m\*(w\*w+h\*h)  iyy = 1.0/12.0\*m\*(h\*h+d\*d)  izz = 1.0/12.0\*m\*(w\*w+d\*d) |

半径r,重さmの球体の場合

|  |
| --- |
| ixx = iyy = izz = 1.0/5.0\*m\*r\*r |

のように計算します。

以下の例のように，linkごとに奥行きや幅を変数宣言しておき，inertiaはマクロで計算できるようにしておくと楽です。

例：base\_linkのマクロ計算例(my\_robo\_description/my\_robo.urdf)

|  |
| --- |
| <!-- base -->  <property name="bd" value="0.330"/>  <property name="bw" value="0.330"/>  <property name="bh" value="0.900"/>  <property name="base\_mass" value="30.0"/>  <property name="base\_ixx" value="${1.0/12.0\*base\_mass\*(bw\*bw+bh\*bh)}"/>  <property name="base\_iyy" value="${1.0/12.0\*base\_mass\*(bh\*bh+bd\*bd)}"/>  <property name="base\_izz" value="${1.0/12.0\*base\_mass\*(bw\*bw+bd\*bd)}"/>  …(省略)  <!-- base\_link -->  <link name="base\_link">  <visual>  <geometry>  <box size="${bd} ${bw} ${bh}"/>  </geometry>  </visual>    <collision>  <geometry>  <box size="${bd} ${bw} ${bh}"/>  </geometry>  </collision>    <inertial>  <origin xyz="0 0 0"/>  <mass value="${base\_mass}"/>  <inertia ixx="${base\_ixx}" ixy="0" ixz="0" iyy="${base\_iyy}" iyz="0" izz="${base\_izz}"/>  </inertial>  </link> |

私が結局解決できなかったのは，base\_footprintの位置です。base\_footprintは回転座標系を示す座標なので回転軸の場所に持ってくるべきなのかと思ったのですが，そうしなくても動くようです。私はbase\_linkの真下，ロボット最下部にある部品の高さに設定しました。この座標を中心として「ロボットを上から見た形状」を定義していくことになります。それは2.4で行います。

2.2 オドメトリを調整する

my\_robo\_control/config/controller.yamlを調整します。

これが非常に大事で，チュートリアルそのままだと回転時に自己位置がずれるのはこのパラメータが間違っているからです。

オドメトリには回転成分と直進成分があり，これらが正しく計算されているかをテストします。

最初に回転成分をテストします。ロボットを起動し，rvizを開き，フレームをodomに設定し，ロボットが提供するレーザースキャンを表示します。トピックの減衰時間を長めに設定（20秒くらい）し，その場で回転してください。

理想的には，回転の2周目でレーザの線は1周目とほぼ一致しているはずです。しかし，回転のずれがある場合は2周目のレーザは1周目のレーザより大きくずれることになります。ずれている場合は，wheel\_separationパラメータを調整してください。

* wheel\_separationはタイヤ中心間の距離（左タイヤの中心と右タイヤ

中心の距離）で，my\_robo.urdfの記述上は0.26で正しいのですが，gazeboで動かす場合はタイヤの外側間の距離でないといけないようです。(なぜそういった仕様なのかは不明)よってチュートリアルで作ったロボットでの正しいwheel\_separationは

|  |
| --- |
| wheel\_separation : 0.31 |

です。しかし，***実機の場合は「タイヤの中心間の距離」が正しいので注意です。***

次のテストは，直進成分のチェックです。ロボットを壁から数メートル離れたところでセットアップします。rvizを開き，回転成分と同じように設定してください。そしてロボットを壁に向かってまっすぐ移動させてください。理想的には壁にそって表示されているレーザは1回のスキャンのように見えるはずです。厚く表示される場合はwheel\_radiusパラメータを調整してください。

以上のテストでオドメトリの調整が完了します。ここまでの設定は，Navigation Stackとは関係ない部分です。

2.3 my\_robo\_2dnav/params/base\_local\_planner.yaml

ここからはNavigation Stackに関係します。まずはmy\_robo\_2dnav/params/base\_local\_planner.yamlを調整します。これはアクセルとブレーキの力をmove\_baseが正しく見積もるために必要な設定です。ここが間違っていると，ロボットが止まりきれずに壁にぶつかったりします。

これを正しく設定するためには，my\_robo\_control/config/controller.yamlの値を確認する必要があります。

以下の記述には接客ロボットに用いた値が入っています。ハードウェア性能に応じて書き換えてください。

my\_robo\_control/config/controller.yaml

|  |
| --- |
| # Velocity and acceleration limits  # Whenever a min\_\* is unspecified, default to -max\_\*  linear:  x:  has\_velocity\_limits : true  max\_velocity : 0.5 # m/s  min\_velocity : -0.5 # m/s  has\_acceleration\_limits: true  max\_acceleration : 0.5 # m/s^2  min\_acceleration : -0.5 # m/s^2  angular:  z:  has\_velocity\_limits : true  max\_velocity : 0.78 # rad/s  min\_velocity : -0.78  has\_acceleration\_limits: true  max\_acceleration : 0.78 # rad/s^2  min\_acceleration : -0.78 |

my\_robo\_2dnav/params/base\_local\_planner.yaml

|  |
| --- |
| TrajectoryPlannerROS:  max\_vel\_x: 0.5  min\_vel\_x: 0.1  max\_vel\_theta: 0.78  min\_vel\_theta: -0.78  min\_in\_place\_vel\_theta: 0.4  acc\_lim\_theta: 0.78  acc\_lim\_x: 0.5  acc\_lim\_y: 0.5  holonomic\_robot: false  yaw\_goal\_tolerance: 0.25  xy\_goal\_tolerance: 0.25  p\_dist\_scale: 1.0  g\_dist\_scale: 0.5  occdist\_scale : 0.1  # heading\_scoring: true  # dwa: true  meter\_scoring: true |

ここで設定されているmax\_vel\_\*\*\*, acc\_lim\_\*\*\*は，controller.yamlで設定した値を超えないようにする必要があります。超えた値を設定しても，その力を出すことはできません。つまり，

controller.yaml：限界性能

base\_local\_planner.yaml：案内時に出したい性能

という感じです。

base\_local\_planner.yamlはcontroller.yamlと同一に設定するのがわかりやすいです。

他にも良く使うパラメータを解説しておきます。

yaw\_goal\_toleranceは目的地にたどり着いたと判定する許容角度誤差です。 xy\_goal\_toleranceは許容座標誤差です。この2つのパラメータに設定した値だけゴール座標からずれている場合でも「ゴールにたどり着いた」と判定されます。甘めに設定しておかないと，なかなかゴールにたどり着いたと判定されません。

2.4 my\_robo\_2dnav/params/costmap\_common\_params.yaml

ここではコストマップに関するパラメータを設定します。障害物に対してどういう挙動をするかを決めるパラメータなので，ここを間違えると障害物にぶつかります。

|  |
| --- |
| obstacle\_range: 3.5  raytrace\_range: 4.0  footprint: [[-0.25, -0.35], [-0.25, 0.35], [0.55, 0.35], [0.55, -0.35]]  inflation\_radius: 0.55  observation\_sources: laser\_scan\_sensor  laser\_scan\_sensor:  sensor\_frame: hokuyo\_link  # sensor\_frame: urg\_laser\_link  data\_type: LaserScan  topic: /scan  observation\_persistence: 0.2  expected\_update\_rate: 0.2  marking: true  clearing: true  inf\_is\_valid: true |

obctacle\_range: ロボットとの距離がobstacle\_range以下のオブジェクトは障害物としてみなし，コストマップに反映します。

raytrace\_range : ロボットとの距離がraytrace\_range以下のオブジェクトが検出された場合，そのオブジェクトの内側のコストマップの障害物がクリアされます。

footprint : ロボットの形状を多角形で指示することができます。footprintの形状に応じてコストマップのdefinitery in collisionとpossibly in collisionの径が決定されます。

inflation\_radius：障害物までの距離がinflation\_radius以下の場所をロボットの中心が絶対に通らないようにします。

接客ロボットではロボットを上から見た形(=footprint)を四角形にしましたが，別に何角形でも大丈夫なので，もっと細かく設定するべきでした。皆さんはそうしてください。

2.5 amclパラメータを調整する

amclパラメータは自己位置推定の精度を上げるために必要です。

amclは次の図のような仕事をします。（パワポから抜粋）

要はオドメトリだけではなく，LRFの情報も使って自己位置推定をするよ，というパッケージです。

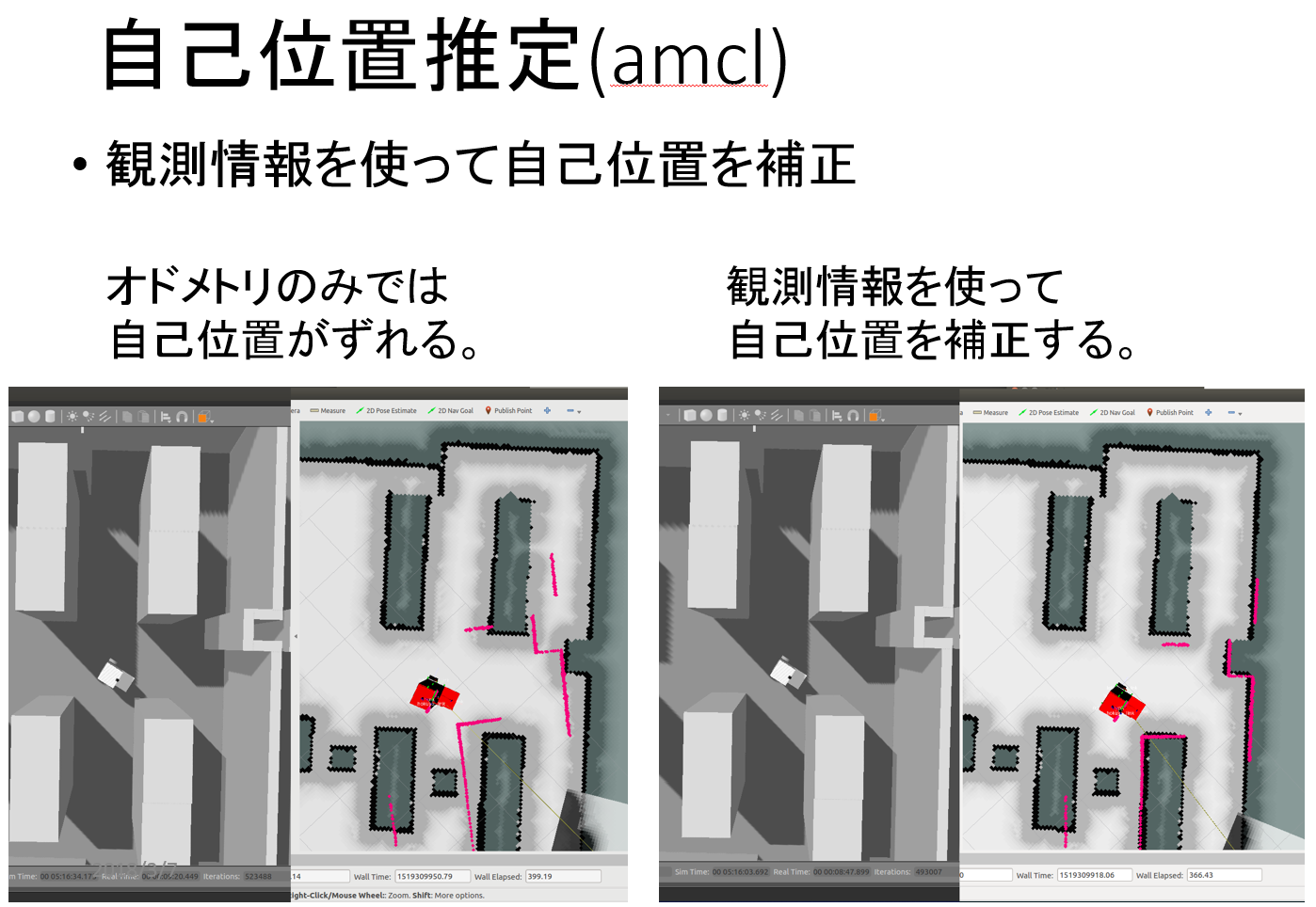


図6-4 オドメトリのみの自己位置推定　図6－5 amclを使った自己位置推定

チュートリアルではlaunchファイルにパラメータが書かれているのですが，私はmove\_baseパラメータと同じようにparamsディレクトリにyamlファイルとしてまとめました。どちらにするかは好みにお任せします。

なお，内容は次のようにしています。

my\_robo\_2dnav/params/amcl\_params.yaml

|  |
| --- |
| # http://wiki.ros.org/amcl#Parameters  # overall filter parameters  min\_particles: 500  max\_particles: 5000  kld\_err: 0.05  kld\_z: 0.99  update\_min\_d: 0.2  update\_min\_a: 0.5  resample\_interval: 1  transform\_tolerance: 0.2  recovery\_alpha\_slow: 0.001  recovery\_alpha\_fast: 0.1  gui\_publish\_rate: 10.0  use\_map\_topic: true  first\_map\_only : false  # laser model parameters  laser\_min\_range: -1.0  laser\_max\_range: -1.0  laser\_max\_beams: 30  laser\_z\_hit: 0.95  laser\_z\_short: 0.05  laser\_z\_max: 0.05  laser\_z\_rand: 0.05  laser\_sigma\_hit: 0.2  laser\_lambda\_short: 0.1  laser\_likelihood\_max\_dist: 2.0  laser\_model\_type: likelihood\_field  # odometry model parameters  odom\_model\_type: diff  odom\_alpha1: 0.2  odom\_alpha2: 0.2  odom\_alpha3: 0.2  odom\_alpha4: 0.2  odom\_alpha5: 0.2  odom\_frame\_id: odom  base\_frame\_id: base\_footprint  global\_frame\_id: map |

重要なのはodom\_alpha1～4です。

これらはそれぞれ

オドメトリの直進成分がオドメトリの直進成分に与える誤差

オドメトリの直進成分がオドメトリの回転成分に与える誤差

オドメトリの回転成分がオドメトリの直進成分に与える誤差

オドメトリの回転成分がオドメトリの回転成分に与える誤差

を表しています。

これらをすべて0にした場合，amclは仕事をしないので，オドメトリが絶対的に正しいものだとみなして動作します。

逆にすべて1にした場合，補正がかかりすぎて，ロボットが実際に進んでいるにもかかわらずrviz上では進まないように見えてしまいます。

実験した結果では0.2くらいがちょうど良かったのでこのように設定しました。

その他のamclパラメータの詳細は

<http://wiki.ros.org/amcl#Parameters>

を参考にしてください。

以上でパラメータ調整は終わりです。

これより細かいチューニングは実際の動作と，本文中に示したURLを参考に行ってください。

３．　**実機で動かすための設定をする（ros\_controlを理解する）**

Navigation Stackとgazeboの仕組みについて解説をします。

gazebo上でのNavigation Stackは**ros\_control**という仕組みを利用して動いています。

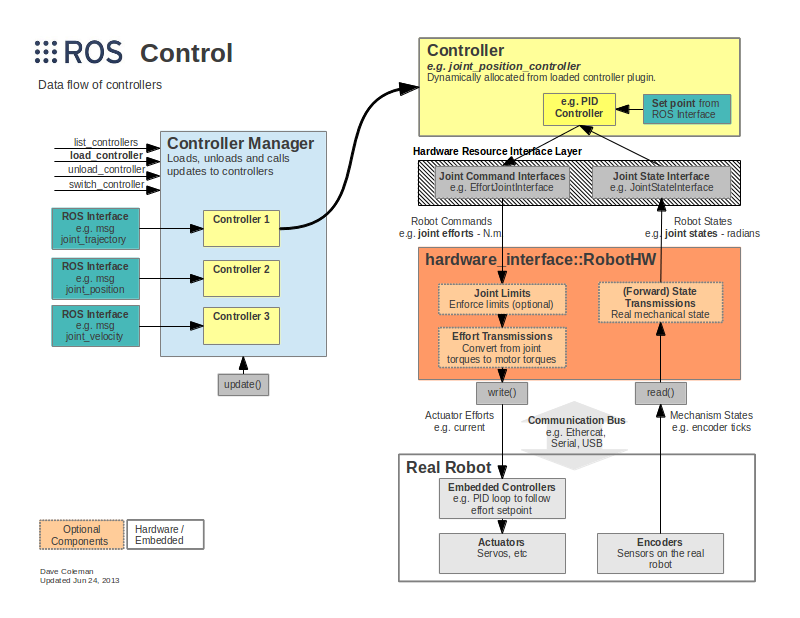
公式サイトからros\_controlの概念図を持ってきます。

図6-6 ros\_control

この**ros\_control**が非常にややこしいのです。私はここがわからなくて実機を動かすために2ヶ月を費やしました。

なぜ難しいかというと，最初に示したNavigation Stackの図（図6－7）と，ros\_controlとの繋がりがわからなかったからです。



図6－7 Navigation Stack 日本語版（再掲）

ros\_controlを用いると図6-7は図6-8のように修正されます。



図6-8 ros\_controlを使った場合のNavigation Stack

つまりNavigation Stackにおけるros\_controlの働きは

・走行指令を受け取って，モータを回転する

・オドメトリを受け取って，トピックとしてpublishする

・ロボットの移動に応じてtf（ロボットの各部品の位置）を計算する

ことです。

**特にtfの計算をしてくれることは助かります**。自分で行うのはとても面倒だからです。urdfファイルを記述することでtfの計算を行ってくれます。

またros\_controlを適切に設定すれば，走行指令以外にも，腕の操作，首の操作もできるので絶対やるべきです。

ros\_controlを解説する記事はネット上にいくつかありますので，参考に載せておきます。[1][2]は難しいです。[3]の記事はやさしく書かれていますのでお勧めです。一度読んで見てください。ros\_controlは便利なので，理解する価値は大いにあります。

[1]Controller と HardwareInterface との間の処理の仕組み（1. ロボットモデルの定義と登録）

<https://qiita.com/MoriKen/items/613635b90f3a98042dc5>

[2]ros\_controls/ros\_controllers の制御の仕組み (position/effort/velocity\_controllers の基礎)

<https://qiita.com/MoriKen/items/78b0ad8c1eae257646dd>

[3]Gazebo + ROS で自分だけのロボットをつくる 5. GazeboとROSの連携

<https://qiita.com/RyodoTanaka/items/6fa7e45f98b55376a95b>

記事を読む前に覚えておいてもらいたいのは，**Navigation Stackのことは「一回忘れる」**ということです。

Navigation Stackはros\_controlの概念図（図6－9）で言うと

ROS Interfaceのe.g. msgの部分に当たるのですが，この図の大部分に関係しませんので，「どこでNavigation Stackが出てくるんだ？」と思いながら読むと混乱します。だから一回忘れてください。

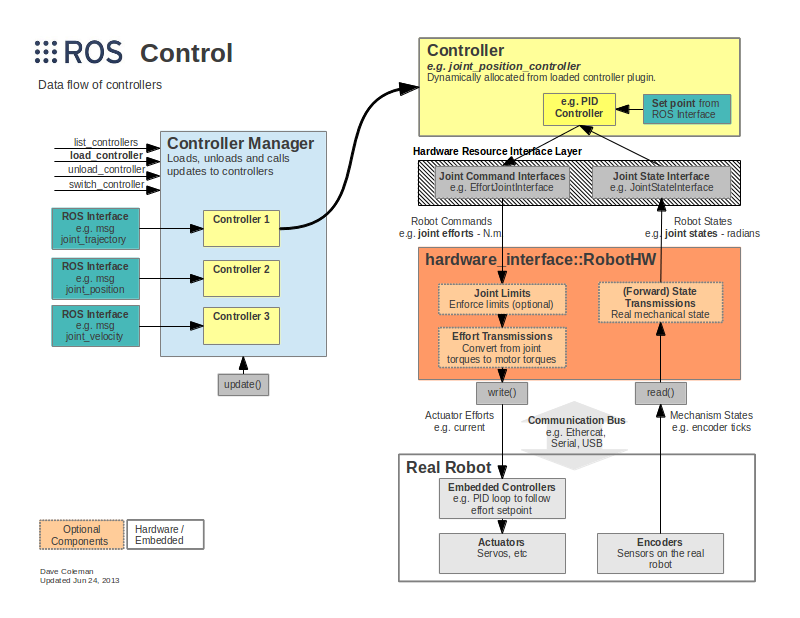


図6－9 ros\_control

しかし，参考URL[1][2][3]を読んでも，実機で動かすための設定をどうすればいいかが分からないと思います。

結論としては，RobotHWというものを作る必要があります。

（ros\_control概念図のhardware\_interface::RobotHWという部分）

gazeboではRobotHWSimというものが動いていて，走行指令を受け取る部分とオドメトリの計算はRobotHWSimの中で行われています。実機で動かす場合はRobotHWSimに代わるものを動かす必要があります。

私はこれをmy\_robo\_controlの中に作りました。headerは次のようなものです。このうち，PositionJointInterfaceは首を回転させるために用意したものなので無視してください。

my\_robo\_control/include/my\_robo\_hw.h

|  |
| --- |
| #include <ros/ros.h>  #include <hardware\_interface/joint\_command\_interface.h>  #include <hardware\_interface/joint\_state\_interface.h>  #include <hardware\_interface/robot\_hw.h>  #include <map>  #include <string>  #include <vector>  class MyRobo : public hardware\_interface::RobotHW  {  public:  MyRobo();  ros::Time getTime() const { return ros::Time::now(); }  ros::Duration getPeriod() const { return ros::Duration(0.01); }  void read(ros::Time time, ros::Duration period);  void write(ros::Time time, ros::Duration period);  int open() const;  void stop() const;  private:  hardware\_interface::JointStateInterface jnt\_state\_interface\_;  hardware\_interface::VelocityJointInterface jnt\_vel\_interface\_;  hardware\_interface::PositionJointInterface jnt\_pos\_interface\_;  double cmd\_[4];  double pos\_[4];  double vel\_[4];  double eff\_[4];  }; |

大事なのはreadメソッドとwriteメソッドです。

・readメソッドで走行指令をモータドライバに送る

・writeメソッドでオドメトリを計算する

ように記述します。

※「readとwriteでやる処理が逆じゃないか？」と思うかもしれませんが，ロボットから見れば移動指令は読み込むべきものであり，オドメトリはユーザに書いて教えてあげるものなので，間違いではありません。

参考にしたソースもそうしてありました。

<https://github.com/open-rdc/icart/blob/indigo-devel/icart_mini_driver/src/icart_mini_driver_node.cpp>

でも，これは最初に示したros\_controlの図とは異なりますので混乱を招いたらごめんなさい。

cppファイルからreadとwriteを一部抜粋します。全文はソースを見てください。

my\_robo\_control/src/my\_robo\_control.cpp

|  |
| --- |
| void MyRobo::read(ros::Time time, ros::Duration period)  {  //if (cmd\_[0] || cmd\_[1])  // ROS\_INFO\_STREAM("Commands for joints: " << cmd\_[0] << ", " << -cmd\_[1]);    int ret = YP\_wheel\_vel(cmd\_[0], -cmd\_[1]);  …// 以降は首回転処理  }  void MyRobo::write(ros::Time time, ros::Duration period)  {  double yp\_vel[2];  yp\_vel[0] = 0;  yp\_vel[1] = 0;  YP\_get\_wheel\_vel(&yp\_vel[0], &yp\_vel[1]);  yp\_vel[1] = -yp\_vel[1];    //if (yp\_vel[0] || yp\_vel[1])  // ROS\_INFO\_STREAM("YPSpur vel: " << yp\_vel[0] << ", " << -yp\_vel[1]);  for (unsigned int i = 0; i < 2; ++i)  {  pos\_[i] += yp\_vel[i] \* getPeriod().toSec();  vel\_[i] = yp\_vel[i];  }  …  } |

readメソッドでYP-Spurの走行命令を，writeメソッドでYP-Spurの読み込み命令を実行しています。

**ypspur\_rosノードとやり取りするのではありません。YP-Spurの命令を直接呼び出してください。**

ちなみにYP-Spurを使うのではなく自作のモータドライバや別のモータドライバを使う場合は，そのモータドライバに書き込むようにソースを変えてください。とにかく**readメソッドでマイコンに書き込んで，writeメソッドでマイコンから読み出す**のです。そうすればモータは動きます。このやり方は，走行系以外でも同じです。

これらをmain関数で次のように呼んでいます。

my\_robo\_control/src/my\_robo\_control.cpp

|  |
| --- |
| while (ros::ok())  {  ros::Time now = myrobo.getTime();  ros::Duration dt = myrobo.getPeriod();  if (YP\_get\_error\_state() == 0)  {  myrobo.read(now, dt);  myrobo.write(now, dt);  }  else  {  ROS\_WARN("Disconnected T-frog driver");  myrobo.stop();    while (myrobo.open() < 0)  {  ROS\_WARN("try to connect T-frog driver");  ros::Duration(1).sleep();  }    ROS\_INFO("connected T-frog driver");  }  cm.update(now, dt);  rate.sleep();  } |

readメソッドとwriteメソッドを一定周期で呼んでいることがわかります。

このソースをcatkin\_makeしてノードを作ります。

そしてgazeboの代わりにlaunchします。rostestではmy\_robo\_controlという名前でノードを作り，次のようにして起動しています。

my\_robo\_control/launch/control\_real.launch

|  |
| --- |
| <launch>  <include file="$(find my\_robo\_control)/launch/control\_base.launch"/>  <node name="my\_robo\_control" pkg="my\_robo\_control" type="my\_robo\_control" output="screen"/>  </launch> |

control\_base.launchは次のようにしています。

my\_robo\_control/launch/control\_real.launch

|  |
| --- |
| <launch>  <arg name="model" default="$(find my\_robo\_description)/my\_robo.urdf"/>    <param name="robot\_description" command="$(find xacro)/xacro.py $(arg model)"/>    <!-- Load joint controller configurations from YAML file to parameter server -->  <rosparam file="$(find my\_robo\_control)/config/controller.yaml" command="load"/>    <!-- load the controllers -->  <node name="controller\_spawner" pkg="controller\_manager"  type="spawner" output="screen"  args="joint\_state\_controller  diff\_drive\_controller  head\_position\_controller  neck\_position\_controller"/>    <!-- convert joint states to TF transforms for rviz, etc -->  <node name="robot\_state\_publisher" pkg="robot\_state\_publisher"  type="robot\_state\_publisher" respawn="false" output="screen"/>  </launch> |

control\_base.launchはチュートリアルそのままで，このlaunchファイルに加えてRobotHWを記述したノード「my\_robo\_control」を動かしています。

実機の起動に必要なノードのlaunchファイルを一部抜粋します。

robot\_launcher/launch/b205\_demo.launch

|  |
| --- |
| <launch>  <!-- setup robot -->  <include file="$(find robot\_launcher)/launch/setup\_robot.launch"/>  <include file="$(find my\_robo\_control)/launch/control\_real.launch"/>  <include file="$(find my\_robo\_2dnav)/launch/move\_base\_B205.launch"/>  （省略）  </launch> |

gazeboの起動に必要だった「シーンとロボットのスポーン」がなくなり，代わりにロボットのセンサ立ち上げ処理がsetup\_robot.launchで行われます。

そしてcontrol\_real.launchでRobotHWを起動します。

最後にmove\_baseの立ち上げをmove\_base\_B205.launchで行っています。

これで実機でNavigation Stackを使うことができます。

シミュレータでロボットを起動する場合のlaunchファイルについて補足します。launchの順番が多少ややこしいのです。

README.mdより

|  |
| --- |
| `roslaunch my\_robo\_description spawn\_world\_b205.launch`  `roslaunch my\_robo\_control control\_base.launch`  `roslaunch my\_robo\_description spawn\_model\_b205.launch`  `roslaunch my\_robo\_2dnav move\_base\_b205.launch`  `rosrun rviz rviz` |

この順番で呼び出す必要があります。そうでなければ起動しません。

また，navigationをやり直すときは

README.mdより

|  |
| --- |
| `move\_base\_b205.launch`を起動しているterminalでCtrl+C  `control\_base.launch`を起動しているterminalをCtrl+C  `roslaunch my\_robo\_description spawn\_model\_b205.launch` でロボットモデルをrespawn  `roslaunch my\_robo\_control control\_base.launch` でコントローラを読み込み直す  `roslaunch my\_robo\_2dnav move\_base\_b205.launch` でnaigation再開 |

この順番で起動し直さないと再開できません。

これはros\_controlがそうなっているからという理由で納得するしかないと思います。

このように細かくlaunchファイルを用意している理由なのですが，これは

「動作を確認してパラメータ調整する」というサイクルを早くまわすためです。

1つのlaunchファイルにして，一旦全て終了して新たに全て起動し直せばもちろんやり直せるのですが，launchファイルを別々にするほうが再開できるまでの時間が短いです。特にgazeboのguiの起動は時間がかかるので，全て終了して起動し直すととても時間がかかります。

guiを一度終了させなくても再開できるように工夫した結果が「別々のlaunchファイルにして一つ一つ起動する」という方法です。

もし面倒であれば1つのlaunchファイルにまとめてください。

**４．FAQ**

Q. gazebo上でロボットの出現する位置を変えたい

A. B205教室であれば，my\_robo\_description/launch/spawn\_model\_b205.launch

の中の，x, y, z, roll, pitch, yawパラメータを変更してください。

これらのパラメータに設定する値は，出現させたい位置にロボットを持ってきて，下図のようにして座標を見て確認してください。(図6-10参照)

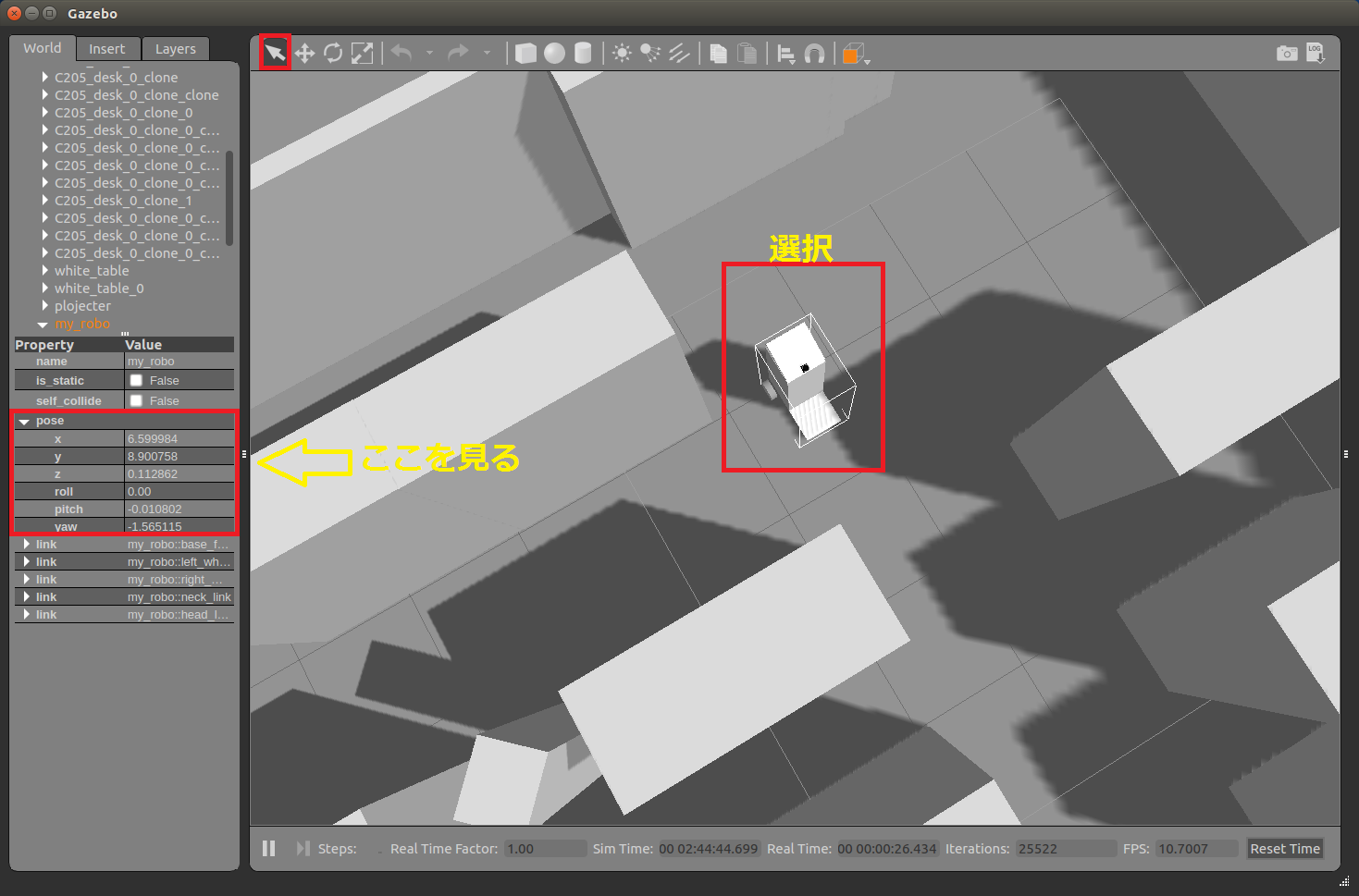


図6-10　gazebo上での座標取得方法

Q. move\_baseを起動したときに現れるロボットの位置を変えたい

A. B205教室であれば，

my\_robo\_2dnav/launch/move\_base\_b205.launch

の中の，x, y, aパラメータを変えてください。

このパラメータは，一度Navigation Stackを起動して，rvizの2D Pose Estimate機能で出現させたい位置にロボットを移動させ，rvizを起動しているターミナルに出力された値を見て設定してください。

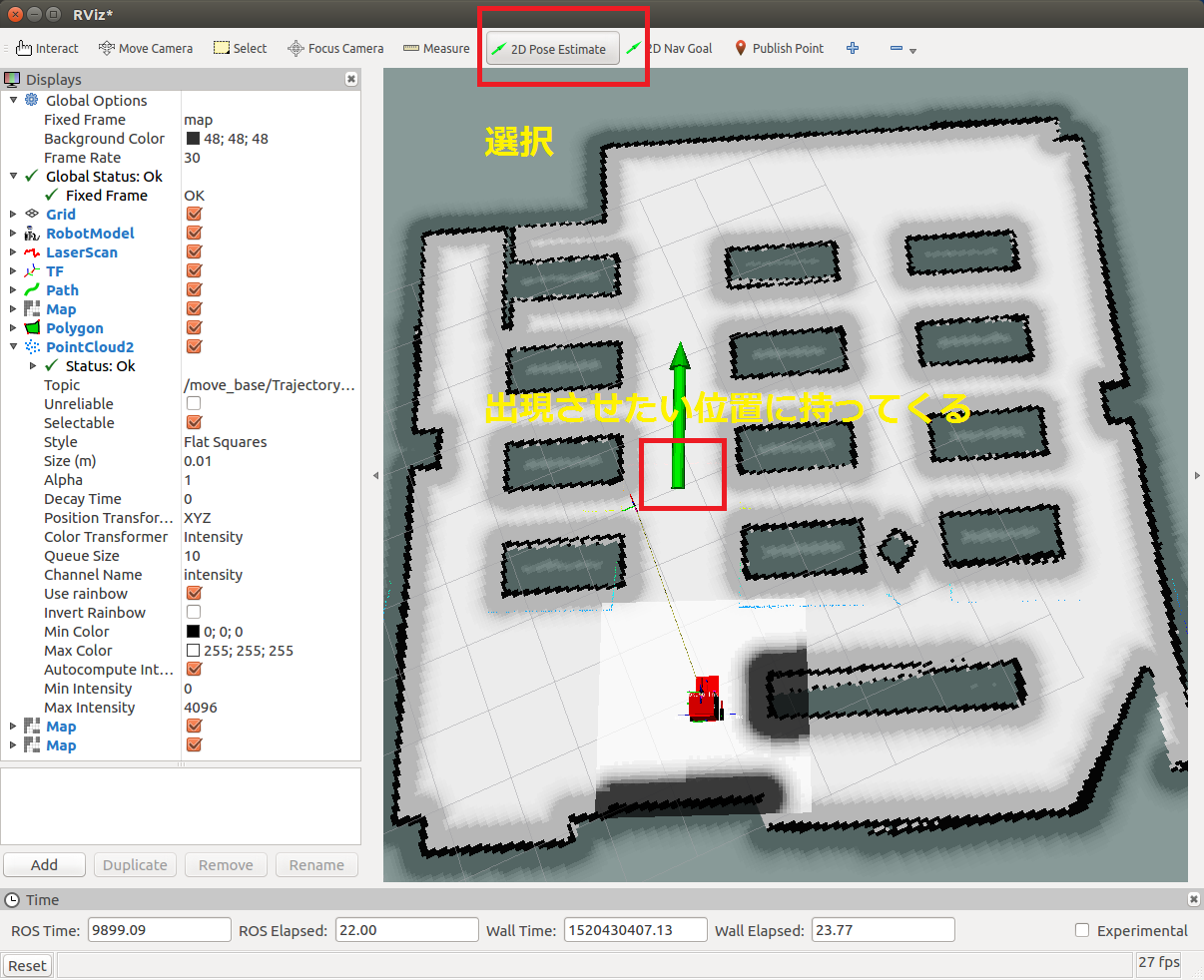


図6-11 rvizでの座標取得方法1

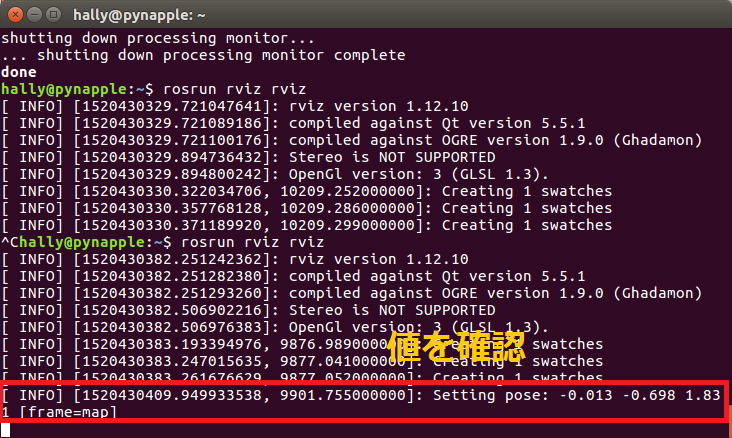


図6-12 rvizでの座標取得方法2

Q. YP-Spurのパラメータ調整方法は？

A. このサイトを見てください。

<http://www.roboken.iit.tsukuba.ac.jp/platform/wiki/yp-spur/parameter-file>

Q. rostestを新しいパソコンにコピーしても動かない

A. 新しいパソコンにおいてROSの初期設定を済ませましたか？

済ませている場合，rostestを動かすのに必要なパッケージが足りていないことが原因だと思われます。

以下のコマンドを実行してください。

|  |
| --- |
| sudo apt-get install ros-kinetic-gazebo\* ros-kinetic-move-base\* ros-kinetic-slam\* ros-kinetic-amcl\* ros-kinetic-map-server\* ros-kinetic-diff-drive-controller ros-kinetic-joint\* ros-kinetic-\*-controllers ros-kinetic-urg-node ros-kinetic-cv-camera ros-kinetic-controller-\* ros-kinetic-control\*  cd ~/catkin\_ws/src  git clone <https://github.com/ros-simulation/gazebo_ros_demos>  cd ~/catkin\_ws  catkin\_make |

Q. デバイスが見つからない

A. ls /devでデバイス名を確認してください。認識されていなければここに名前が出てこないので，USB接続がされているか確認してください。USBハブが古い場合，USB端子を逆向きに挿している可能性も考えられます。

Q. デバイスを挿す順番でデバイス名が変わる

A. udevで調べてください。

Q. 新しいマップを作りたい

A. マップを作りたい場所の寸法を測ってください（床のタイル何個分という情報を使うとわかりやすいです）

そしてその通りにgazebo上に3Dオブジェクトを配置してください。

ぴったりな3Dオブジェクトがない場合は自分で作ってください。

gazebo→Edit→Building Editorで作れます。

作ったら，シーンファイルを保存してください。

保存して出来上がったシーンファイルをrostest/my\_robo\_description/worldに置き，launchファイルから起動するようにしてください。rostestでシーンファイルfood\_space\_ver3を起動する例を示します。

my\_robo\_description/launch/spawn\_world\_food.launch

|  |
| --- |
| <launch>  <include file="$(find my\_robo\_description)/launch/spawn\_world.launch">  <arg name="world" value="food\_space\_ver3"/>  </include>  </launch> |

次に，urdfファイルを編集してください。これによりロボットの首がまっすぐになります。（ロボットの首は通常時は下に20°に傾いています）

my\_robo\_description/my\_robo.urdf

|  |
| --- |
| <!-- head -->  <property name="hd" value="0.310"/>  <property name="hw" value="0.330"/>  <property name="hh" value="0.03"/>  <property name="hx" value="0"/>  <property name="hy" value="0"/>  <property name="hz" value="${nh/2.0+hh/2.0}"/>  <property name="hp" value="${M\_PI/9.0}"/> **ここをコメントにして**  <!--<property name="hp" value="0"/>-->　**このコメントをはずす**  <property name="head\_mass" value="4.0"/>  <property name="head\_ixx" value="${1.0/12.0\*head\_mass\*(hw\*hw+hh\*hh)}"/>  <property name="head\_iyy" value="${1.0/12.0\*head\_mass\*(hh\*hh+hd\*hd)}"/>  <property name="head\_izz" value="${1.0/12.0\*head\_mass\*(hw\*hw+hd\*hd)}"/> |

そして，作ったgazebo空間にロボットをスポーンするためのlaunchファイルを作ってください。

rostestでfood\_space\_ver3にロボットをスポーンする例を示します。

|  |
| --- |
| <launch>  <!-- spawn robot -->  <include file="$(find my\_robo\_description)/launch/spawn\_model.launch">  <arg name="x" value="-19.0"/>  <arg name="y" value="-0.56"/>  <arg name="z" value="0.11"/>  <arg name="roll" value="0.0"/>  <arg name="pitch" value="0.01"/>  <arg name="yaw" value="-1.585"/>  </include>  </launch> |

シーンとロボットをスポーンし，コントローラを起動してください。

food\_space\_ver3にスポーンするためのコマンドは次のようになります。

（spawn\_world\_food.launchとspawn\_model\_food.launchは自分で作ったlaunchファイルに置き換えてください）

|  |
| --- |
| roslaunch my\_robo\_description spawn\_world\_food.launch  roslaunch my\_robo\_control control\_base.launch  roslaunch my\_robo\_description spawn\_model\_food.launch |

そのあと，次のコマンドを実行してください。

|  |
| --- |
| rosrun gmapping slam\_gmapping |

別のターミナルで，次のコマンドを実行してください。

|  |
| --- |
| roslaunch my\_robo\_control run.launch |

経過はrvizで確認してください。地図が出来上がる様子が確認できます。

地図が全部出来上がるまで走らせたら，別のターミナルを立ち上げて，

|  |
| --- |
| rosrun map\_server map\_saver |

を実行してください。このコマンドを実行したディレクトリに，map.pgmとmap.yamlができあがります。これで地図の完成です。

Q. プログラムを書くのが面倒

A. あなたがやろうとしていることは誰かやろうとしています。パッケージがすでにあるかもしれません。ヘッダーファイルとして提供されているかもしれません。まずはそれを探しましょう。「作らないソフトウェア」がROSです！

Q. それでもわからないことがある

A. まずはROS Wikiを調べてください。解決率は3割から4割だと思います。英語は翻訳機能をうまく使ってください。日本語と英語のページを両方開くのがお勧めです。